THE CONSISTENT COMPARISON PROBLEM IN N-VERSION SOFTWARE

Susan S. Brilliant

John C. Knight

Nancy G. Leveson

Computer Science Report No. TR-86-17b
June 25, 1986
Revised August 5, 1986

Submitted to *IEEE Transactions on Software Engineering*.

THE CONSISTENT COMPARISON PROBLEM

IN *N*-VERSION SOFTWARE

Susan S. Brilliant

John C. Knight

Nancy G. Leveson

Affiliation Of Authors

Susan S. Brilliant   John C. Knight

Department of Computer Science

University of Virginia

Charlottesville

Virginia, 22903

Nancy G. Leveson

Department of Computer Science

University of California

Irvine

California, 92717

Financial Acknowledgement

This work was supported in part by NASA under grant numbers NAG-1-242, NAG-1-605 and NAG-1-606, in part by NSF grant DCR 8406532, and in part by MICRO grants cofunded by the state of California, Hughes Aircraft Company, and TRW.

Index Terms

Design diversity, fault tolerant software, multi-version programming, *N*-version programming, software reliability,

Address For Correspondence

John C. Knight

Department of Computer Science

University of Virginia

Charlottesville

Virginia, 22903

Abstract

We have identified a difficulty in the implementation of *N*-version programming. The problem, which we call the *Consistent Comparison Problem,* arises for applications in which decisions are based on the results of comparisons of finite-precision numbers. We show that when versions make comparisons involving the results of finite-precision calculations, it is impossible to guarantee the consistency of their results. It is therefore possible that correct versions may arrive at completely different outputs for an application which does not apparently have multiple correct solutions. If this problem is not dealt with explicitly, an *N*-version system may be unable to reach a consensus even when none of its component versions fail.

I INTRODUCTION

Multi-version or *N*-version programming [2] has been proposed as a method of providing fault tolerance in software. The approach requires the separate, independent preparation of multiple (*i.e.* "*N*") versions of a piece of software for some application. These versions are executed in parallel in the application environment; each receives identical inputs and each produces its version of the required outputs. The outputs are collected by a voter and, in principle, they should all be the same. In practice there may be some disagreement. If this occurs, the results of the majority (if there is one) are assumed to be the correct output, and this is the output used by the system.

It is, of course, possible that there is no majority result. In most practical systems, this possibility is handled by designing a "backup" or fail-safe system which takes over in case of failure to tolerate a fault (or faults). In simple systems, this may involve gracefully shutting down the computer system. In other more complex and critical applications, more sophisticated fail-safe designs may be required. In any case the *N*-version system has failed, resulting in at least a partial degradation in service and perhaps more serious consequences.

In the process of performing a large-scale experiment in *N*-version programming [5-7], a significant difficulty in the implementation of such systems has come to light which affects the ability of an *N*-version system to reach a consensus. The problem derives from the use of finite-precision arithmetic and the uncertainty that arises in making comparisons with finite-precision numbers. We refer to this problem as the *Consistent Comparison Problem.*

II THE CONSISTENT COMPARISON PROBLEM

When finite-precision arithmetic is used, the result of a sequence of computations depends on the order of the computations and the particular arithmetic algorithms used by the hardware. The issue that this raises for *N*-version systems is best illustrated by an example.

Any realistic application will require various comparisons to be made during the computation, and some of these will be based on parameters of the application as defined in the specification. For example, in a control system, the specification may require that the actions of the system depend upon quantities such as temperature and pressure that are measured by sensors. The values of temperature and pressure used by a program may be the result of extensive computation on sensor measurements. Once these values are computed, however, the actions required at temperatures below 100°C may be very different from actions required at temperatures above 100°C, and, similarly, the actions required may differ according to whether the pressure is above or below 15psi.

Now consider such an application implemented using a *3*-version software system. Suppose that at some point within the computation, an intermediate quantity has to be compared with some application-specific constant C_1 in order to determine the required processing. As a result of the various limitations of finite-precision arithmetic, it is quite likely that the three versions will have slightly different values for the computed intermediate quantity, say R_1, R_2, and R_3. If the R_i are very close to C_1 then it possible that their relationships to C_1 are different. Suppose that R_1 and R_2 are less than C_1 and R_3 is greater than C_1. If the versions base their execution flow on these relationships, then two will follow one path and the third a different path. These differences might cause the third version to send to the voter a final output that differs from the other two.

It could be argued that this slight difference is irrelevant because at least two versions will agree, and, since the R_i are very close to C_1, either of the two possible outputs that would result from the use of the R_i would be satisfactory for the application. If only a single comparison is involved then this is correct. However, suppose that a second decision point is required by the application and that the constant involved is C_2. Only two versions will arrive at the decision point involving C_2 having made the same decision about C_1. Now suppose that the values produced by these two versions are on opposite sides of C_2. If the versions base their control flow on this comparison with C_2, then again their behavior will differ. The effect of the two comparisons, one with C_1 and one with C_2, is that the three versions might obtain three different final outputs, all of which may well have been acceptable to the application, but a vote might not be possible. The situation is shown graphically in figure 1. Despite the fact that this example is expressed in terms of comparison with C_1 and C_2, the order is irrelevant. In fact, since the versions were prepared independently, different orders are likely.

Although an application may seem to have only a single solution (unlike a shortest-path-through-a-network computation, for example, that sometimes has more than one), the inconsistency in comparisons leads to the possibility that multiple correct outputs may be produced for a given input. This is an unexpected variant of the well-known "multiple correct results" problem in N-version programming [1]. The problem does not lie in the application itself, however, but in the specification. Specifications do not (and probably cannot) describe required results down to the bit level for every computation and every input to every computation. This level of detail is necessary, however, if the specification is to describe a function, *i.e.* one and only one output is valid for every input.

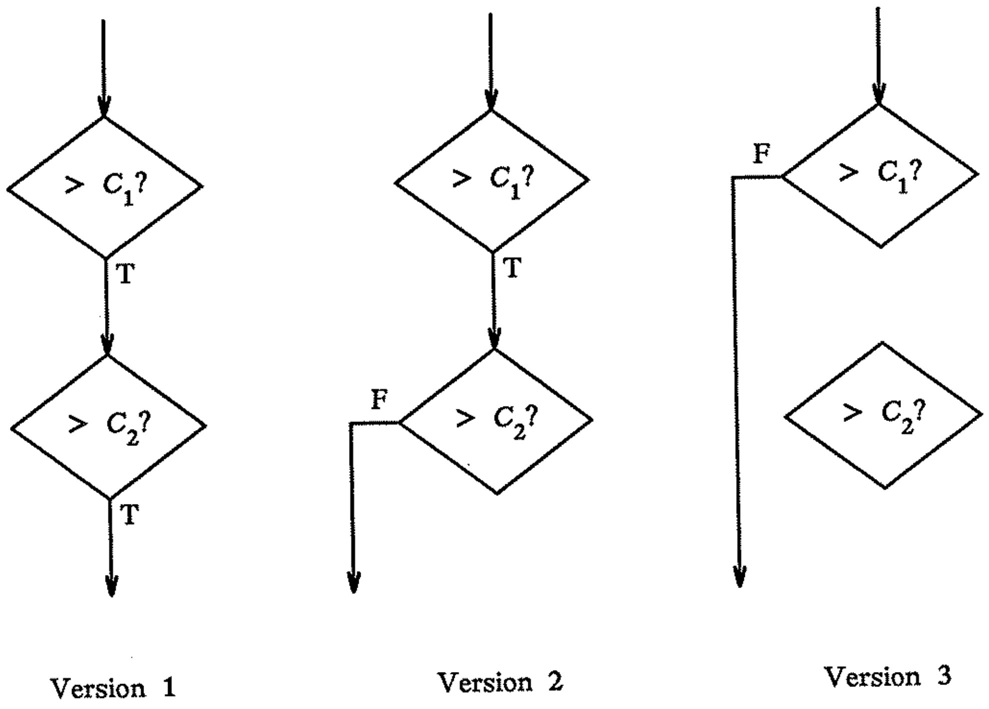

Figure 1 - Three Version Disagreement

In summary, the issue is that multiple software versions might arrive at different conclusions because they take different paths based on comparisons that are required by the specification. The reason for the different paths is the inevitable difference in computed values within the versions due to finite-precision arithmetic and the diversity in the algorithms. In the example based on temperatures and pressures, the effect may be that the temperatures computed internally by the versions straddle 100°C and the computed pressures straddle 15psi. The Consistent Comparison Problem is to avoid this situation.

Consistent Comparison Problem

Suppose that each of N programs has computed a value. Assuming that the computed values differ by less than ϵ ($\epsilon > 0$) and that the programs do not communicate, the programs must obtain the same order relationship when comparing their computed value with any given constant.

It is important to understand that the Consistent Comparison Problem is not related to the well-known problem in which the voter in an N-version system has to deal with slightly different numeric values due to finite precision arithmetic. This latter problem has received considerable attention, and there have been a number of proposed solutions such as inexact voting [4]. The Consistent Comparison Problem derives from the need for versions to make isolated comparisons and can lead to output values that are completely different rather than values that differ merely by a small tolerance.

III NON-SOLUTIONS

A solution to the Consistent Comparison Problem requires that if comparison is needed with a constant C obtained from the specification, a given version must conclude that its value is, say, greater than C if and only if all correct versions will make the same decision given their individual roundoff and truncation errors. This must occur no matter in what order each version chooses to make the comparisons.

It might seem that using approximate rather than exact comparison within each version would solve the problem. An approximate comparison algorithm regards two numbers as equal if they differ by less than some tolerance δ [8]. If a computed value is to be compared with a constant C, approximate comparison requires performing comparisons with numbers that differ from C by the tolerance, δ. For example, it can be concluded that the computed value is greater than C only if it is greater than $C + \delta$. Unfortunately, approximate comparison is not a solution because the problem immediately arises again, this time with $C + \delta$ rather than C. Two programs may compute values that are arbitrarily close to each other but have different order relationships with $C + \delta$, just as with C.

In fact, for two versions to obtain the same order relationship when comparing their computed values with a constant requires that the two computed values be identical. To solve the Consistent Comparison Problem, an algorithm is needed that can be applied independently by each correct version to transform its computed value to the same representation as all other correct versions. It is not sufficient that the values be very close to each other since, no matter how close they are, their relationships to the constant may still be different. However, it is not necessary that all precision be retained and so an approach using rounding or truncation would be satisfactory. It is important to keep in mind that the algorithm must operate with a single value and no communication between versions to exchange values can occur since these are values produced by intermediate computation, not final outputs. Unfortunately, as the following theorem shows, there is no such algorithm, and so there is no solution to the Consistent Comparison Problem.

Theorem

Other than the trivial mapping to a predefined constant, no algorithm exists which, when applied independently to each of two n-bit integers that differ by less than 2^k, will map them to the same m-bit representation $(m + k \leqslant n)$.

Proof

Suppose such an algorithm exists. Consider the case $k = 1$ and the set of values that can be represented in n bits, *i.e.* the integers in the range 0 to $2^n - 1$. The algorithm, when applied to each of two integers that differ by one, *i.e.* adjacent integers, will cause them to be reduced to the same m-bit representation. Thus, 0 and 1 must be mapped to the same representation. However, 1 must also be reduced to the same representation as 2, 2 to the same representation as 3, and in general i to the same representation as $i + 1$. Thus the algorithm must reduce all of the values to the same representation, *i.e.* the algorithm can only be the trivial mapping to any fixed binary representation, and the information content of the values is lost. A similar argument can be made for any k, and so the original assumption is wrong. ■

It is easy to be fooled into thinking that an approach such as rounding, for example, solves the problem. As the theorem shows, however, it does not. We note that this problem is not limited to comparison with constants. It arises with any comparison involving floating point values. For example, if two computed quantities F_1 and F_2 have to be compared, this is equivalent to comparing their difference with zero, a constant.

IV IMPRACTICAL AVOIDANCE TECHNIQUES

Since no solution exists to the Consistent Comparison Problem, we turn our attention to the problem of avoiding its effects. In this section we discuss avoidance techniques that are generally impractical although they might be used successfully in specific applications. In Section VI we consider a practical avoidance technique.

Random Selection

When an N-version system fails to reach a consensus because of inconsistent comparison, all N outputs are, in principle, acceptable to the application, and it might seem that the voter could be modified to select one of the outputs at random. However, an N-version system may also be unable to reach a consensus because a majority of the versions fail. These two cases are indistinguishable, and so modifying the voter in an attempt to deal with inconsistent comparison will also operate when the disagreement is due to failures. This approach is not always satisfactory and could be very dangerous since the selected output could lead to an unsafe course of action.

Exact Arithmetic

Since the difficulty seems to arise from finite-precision arithmetic, a tempting approach is to require some form of exact arithmetic in the versions. However, in addition to being impractical to use for most applications, exact arithmetic will not work in general because many algorithms are themselves capable of producing only approximate solutions. For example, consider the numerical solution of a differential equation which has no closed-form solution. The solution that is obtained is based on a discrete approximation to the equation and different algorithms will very likely use different discretizations. Different solutions may be obtained even though exact arithmetic is used in the calculations.

Cross-Check Points

It might be argued that a way to avoid inconsistency in comparisons would be to establish "cross-check" points [2] to force agreement among the versions on their floating point values before any comparisons are made that involve these values. At each cross-check point, each version would supply its computed result to a cross-check voter. The cross-check voter would select a single value (perhaps the median of the individual results) which would then be used by all of the versions in making the required comparisons. We note that this is not the purpose for which cross-check points were originally intended. They were introduced to allow recovery from software faults at the subsystem level, and were not intended to be applied at this micro level.

Clearly, in principle, this avoids the problem, but various practical difficulties arise. First, if cross-check points are needed to avoid inconsistent comparison, they must be required in the specification. This means that the *order* of cross-check points must be specified also since otherwise the versions will deadlock. Requiring that the order of events in each program version be specified is a serious limitation on diversity in a situation where diversity is being sought. Ideally, no information about the design should be included in the requirements.

In fact, in many applications, this requirement would limit diversity among versions to the point of absurdity. For example, the launch interceptor problem used in our N-version experiment [5] requires that the program determine whether fifteen "launch conditions" are satisfied. Each launch condition requires the determination of the existence of certain geometric relationships among subsets of up to 100 data points representing points in the plane. One of the launch conditions, for example, is satisfied if three data points can be contained within a circle of radius R, where R is a parameter of the application.

Some of the versions written for the experiment were structured to consider each of the fifteen launch conditions in turn, and, during the evaluation of each condition, the individual data points were considered. Others were structured to work through the sets of data points, considering for each the launch conditions that could be satisfied by the data points. If cross-check points were used to avoid inconsistent comparison, it would be necessary to require one of these two basic program structures. The order in which the launch conditions are considered and the order in which the individual data points are examined would have to be established as well.

Finally we note the inefficiency that cross-check points introduce. In highly reliable systems, it is important to take account of Byzantine faults [9]. If the different versions are executing on different processors, it will be necessary for the cross-check points to establish the values to be voted using Byzantine agreement. In addition, since all versions must reach a cross-check point before a vote is possible, a vote cannot occur until the slowest version has arrived. Thus the execution time of the system will be the sum of the execution times of the slowest version between each pair of cross-check points.

Certainly there is a class of applications that can use cross-check points to solve the Consistent Comparison Problem, but the problem of overspecification leading to lack of design diversity is a serious one.

V THE CONSEQUENCES OF THE PROBLEM

It is necessary to understand the effects of the Consistent Comparison Problem before seeking practical methods for avoiding them. The immediate effect of inconsistent comparison is that a consensus might not be possible. The extent of the damage varies with the application and has a substantial impact on the effectiveness of measures designed to cope with the damage. The major characteristic that needs to be considered is whether or not the application has state information that is maintained from frame to frame, i.e. "history".

Some simple control systems have no history. They compute their outputs for any given frame solely from constants and the inputs for that frame. If the inputs are changing, it is extremely unlikely that a situation in which no consensus is possible would last for more than a short time. After a brief period, the inputs will change and leave the region of difficulty. Subsequent comparisons will be consistent among the versions because the values used for comparison will be sufficiently different. The effects of the Consistent Comparison Problem in such systems are transient.

For systems with history, the situation is much more complex. Since such systems maintain internal state information over time, an unfortunate consequence of inconsistent comparison is that failure to reach a consensus might be accompanied by differences in the internal state information among the versions. The duration of internal state differences varies among applications.

In some applications, the state information is revised as time passes and, once the inputs have changed so that comparisons are again consistent, the versions may revise their states to be consistent as well. In this case, the entire system is once again consistent and operation can safely proceed. An example of this type of application is an avionics system in which the mode of flight is maintained as internal state information. If this flight mode is determined by height above ground, for example, then if a measurement is taken that is close to the value at which the mode is changed, different versions might reach different conclusions about which mode to enter. However, it is likely that this situation will be corrected rapidly if the versions continue to monitor the height sensor. We will refer to such systems as having *convergent* states.

For other applications, some state information is determined and then never reevaluated. An example of this is sensor processing where one version may determine that a sensor has failed and subsequently ignore it. Other versions may not make the same decision at the same point in time, and, depending on subsequent sensor behavior, may never conclude that the sensor has failed. In this case, although the inputs change, versions may continue to arrive at different correct outputs long after comparisons become consistent because the sets of state information maintained by the individual versions are not the same. We will refer to these systems as having *non-convergent* states.

VI A PRACTICAL AVOIDANCE TECHNIQUE

We now turn our attention to a practical avoidance technique. This approach requires enhancements to be made to the implementation of an N-version system.

Systems With No History

For systems without history, the only effect of inconsistent comparisons will be a transient inability to reach a consensus. An approach appropriate for this case using "confidence signals" has been suggested by Bishop [3]. Although any particular version cannot have access to the results of comparisons performed by other versions, it can determine that the values it used for comparison were sufficiently close that a problem might have arisen. If this occurs, a solution that can be considered is to require that each version signal to the final voter that a comparison could have been inconsistent. This would allow the voter to distinguish between failures to reach consensus due to inconsistent comparisons, and those due to version failures, so that it would be safe to select an output randomly in cases of inconsistent comparison. This approach requires fairly extensive modifications to the system structure. Each output would have to be supplemented by the required confidence signal, and the voter would have to be modified to take these signals into account.

Systems With Convergent States

For systems having convergent states, inconsistent comparisons may cause a temporary discrepancy among version states. If a confidence signal approach is to be used, each version must maintain confidence information as part of its state. If a part of its state information is based on a comparison that is subject to doubt, then the version must indicate "no confidence" in all of the results it sends to the voter until the state is reevaluated. The time required to reevaluate the state is application dependent. We note that during this period the system is not fault tolerant, and the time to reevaluate the state may be unacceptably long.

Systems With Non-Convergent States

Once the versions in a system with non-convergent states acquire different states, the inconsistency may persist indefinitely. Although no version has failed, the versions may continue to produce different outputs, and, in the worst case, the *N*-version system may never again reach a consensus on a vote. We see no simple avoidance technique that can be used in this case. The only practical approach in systems of this type seems to be to revert to the backup system mentioned in Section I.

VII CONCLUSION

The Consistent Comparison Problem arises whenever a quantity used in a comparison is the product of inexact arithmetic. The problem occurs even when all software versions are correct. It results from rounding errors, not software faults, and so an *N*-version system built from "perfect" versions may have a non-zero probability of being unable to reach a consensus.

This paper has presented some possible solutions for particular types of simple applications, but has also shown that no general, practical solution to the Consistent Comparison Problem exists. This result is important because if no steps are taken to avoid it, the Consistent Comparison Problem may cause failures to occur that would not have occurred in non-fault-tolerant systems. In general, there is no way of estimating the probability of such failures. The failure probability will depend heavily on the application and its implementation. Although this failure probability may be small, such causes of failure need to be taken into account in estimating the reliability of N-version software.

ACKNOWLEDGEMENTS

We are pleased to acknowledge helpful discussions about the Consistent Comparison Problem with Paul Ammann, Peter Bishop, Bruce Chartres, James Ortega, Brian Randell, and Dana Richards. This work was supported in part by NASA under grant numbers NAG-1-242, NAG-1-605 and NAG-1-606, in part by NSF grant DCR 8406532, and in part by MICRO grants cofunded by the state of California, Hughes Aircraft Company, and TRW.

REFERENCES

[1] T. Anderson and P. A. Lee, *Fault Tolerance, Principles and Practice*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[2] A. Avizienis and L. Chen, On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution, *Proceedings of COMPSAC 77*, November 1977, pp. 149-155.

[3] P. Bishop, personal communication, June 1986.

[4] L. Chen and A. Avizienis, N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation, *Digest FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, Tolouse, France, June 1978, pp. 3-9.

[5] J. C. Knight, N. G. Leveson and L. D. St. Jean, A Large Scale Experiment in N-Version Programming, *Digest FTCS-15: Fifteenth Annual International Conference on Fault Tolerant Computing*, Ann Arbor, Michigan, June 1985.

[6] J. C. Knight and N. G. Leveson, *Digest FTCS-16: Sixteenth Annual International Conference on Fault Tolerant Computing*, Vienna, Austria, June 1986.

[7] J. C. Knight and N. G. Leveson, An Experimental Evaluation of the Assumption of Independence in Multiversion Programming, *IEEE Transactions on Software Engineering*, January 1986, pp. 96-109.

[8] D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, Massachusetts, 1969.

[9] L. Lamport, R. Shostak and M. Pease, The Byzantine Generals Problem, *ACM Transactions on Programming Languages and Systems*, July 1982, pp. 382-401.